連載 2

割り算回路設計，あの手この手

デバイスの記事

関連データ

非回復法による除算回路の設計

鈴木昌治

今回は，非回復法というアルゴリズムを用いた除算回路の設計法を説明する．回復法とは異なり，非回復法では部分余が除数より小さい場合でも減算を行う．その結果，部分余が負になった場合は次のサイクルで除数を加算する．つまり，符号付きの演算が行える．なお，本稿で紹介した回路のHDLソースは，本誌付属のCD-ROMに収録されている．（編集部）

前回（本誌2005年8月号，pp.109-117）は回復法をベースとして除算回路の基本についてお話ししました．そのとき紹介した回路はあくまでも基本的なものであり，動作速度の改善や回路規模の抑制などに不利な面があり，また無符号の引き数しか扱えない構造でした．

今回は非回復法を取り上げます．ここでは，取り扱う引き数も符号付きへと発展させ，高速化へのアプローチとしましょう．

1 引きすぎてもよい「非回復法」

非回復法は，基本的な構造そのものは回復法と似ており，上位側から商を確定していき，徐々に余を0に近づけていくという方法です．もっとも大きな違いは，「引きすぎてはならない」という回復法特有の制約を取り払っていることです．回復法に見られた筆算の延長上では，ある意味「非常識」なことなのですが，どうしてそのようなことが可能なのでしょうか．

注1：本連載において，dは除数を，zは被除数を，qは商を，rは余を表すものとする．

●「部分余が負になったら加算」というルールを追加

回復法において，iサイクル目でシフト・アップ済みの部分余をpr_iとします．このサイクルで減算の条件（$pr_i \geqq d$[注1]）が満たされれば，式(1)が実行されます．

$$sub_result_j = pr_j - d \quad \cdots\cdots(1)$$

逆に，減算の条件が満たされなかったとすると，減算は実行されず（回復），pr_iは次のサイクルに持ち越しとなります．次のサイクルで減算が実行されると，式(2)が実行されます．

$$sub_result_{j+1} = 2pr_j - d \quad \cdots\cdots(2)$$

この2サイクルにおける商は，$01_2 = 0.5_{10}$です．

ここで，減算の条件が満たされていないにもかかわらず，iサイクル目で減算を実行してしまった（非回復）としましょう．すると，次のサイクルで減算を実行した場合に，

$$\begin{aligned} sub_result_{j+1} &= 2\left(pr_j - d\right) - d \\ &= 2pr_j - 3d \end{aligned} \quad \cdots\cdots(3)$$

となり，式(2)とは異なる誤った結果となります．

このとき，加算が可能であるとしたらどうでしょうか．つまり，減算の条件が満たされていないということは式(1)を実行すると負になりますが，そこに「部分余が負のときは除数を加算する」というルールを適用するわけです．このルールを適用したものが式(4)であり，答えも式(2)と一致します．

$$\begin{aligned} sub_result_{j+1} &= 2\left(pr_j - d\right) + d \\ &= 2pr_j - d \end{aligned} \quad \cdots\cdots(4)$$

このとき部分商としては－1を立てて，加算という逆の

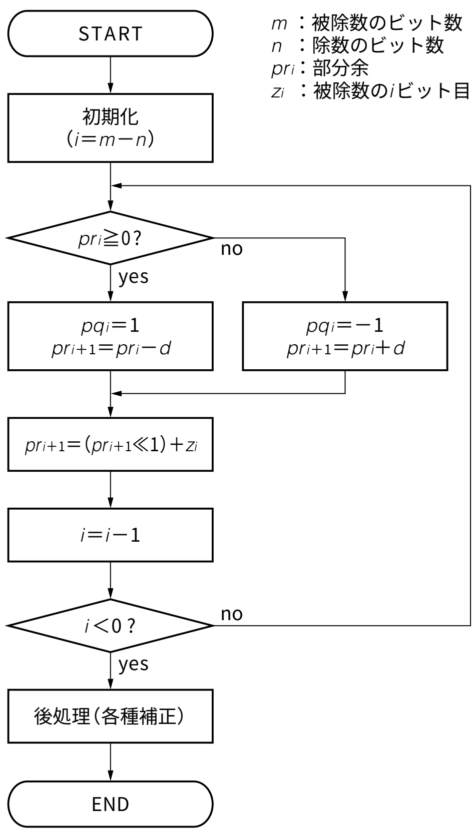
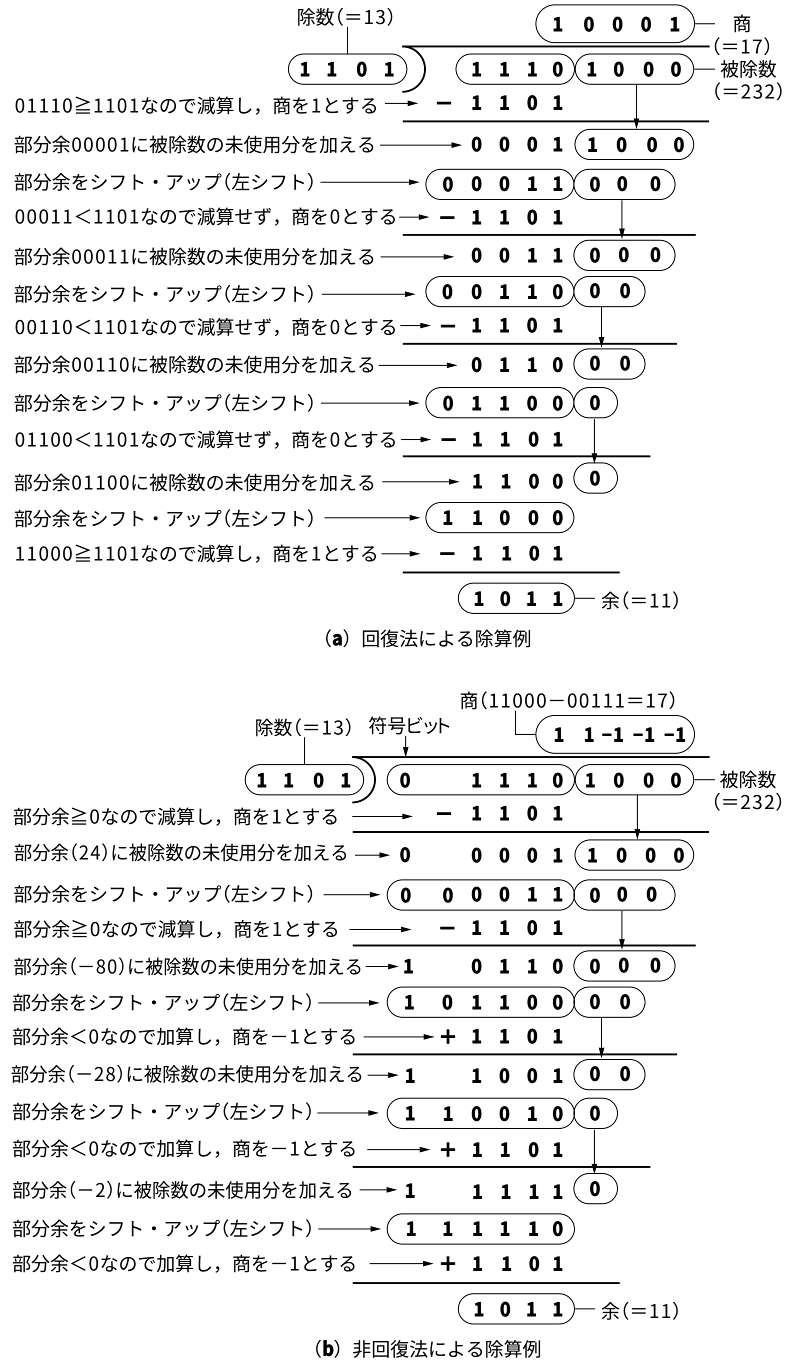


図1　非回復法による除算アルゴリズム

非回復法による除算アルゴリズムをフローチャートで示す．



▶ 図2
除算の例

比較のため，(a)に回復法による除算を示す．(b)の非回復法では，回復法の場合と異なり，部分余が一時的に負の値をとることに注意．なお，前回は，算数で習った筆算の延長で説明したため，けたを不動のものとして扱ったが，論理回路を考えるうえでは図のように左詰めにするほうがより現実的といえる．

操作を実行した結果を残します．この2サイクルにおける商は，$(10-01)_2=(1-0.5)_{10}=0.5_{10}$で，商としても一致していることがわかります．

こうして，回復法における「iサイクル目の減算のスキップと$i+1$サイクル目の減算実行」で得られる結果と同じものが，「iサイクル目の減算実行と$i+1$サイクル目の加算実行」で得られたわけです．これが非回復法です．iサイクル目における誤った(？)減算を$i+1$サイクル目で加算することで回復しているので，「非回復」というのもおかしいのですが，回復のステップが1サイクルで閉じていない，というところが回復法との大きな違いといえるでしょう．

● 操作は異なるが回復法と同じ結果が得られる

フローチャートを用いた非回復法のアルゴリズムを**図1**に示します．また，**図2**にその例を示します．ここで，一つ注意が必要です．それは，非回復法では部分余が一時的

に負になることがありうるということです．前回は筆算の延長で説明した関係から，けたを不動のものとして扱いました．しかし，この方法では符号ビットの位置がわかりにくくなるので，今回は「シフト・アップ」という場合，**図2**のように左に詰めていくことにします．

以下に，非回復法の演算過程を説明します（回復法については，前回を参照）．

最初の1サイクルは$232-13\times2^4\geqq0$となり減算可能なので，商には‘1’が立ちます．これは回復法と同じ操作です．しかし，次のサイクルでは24（部分余）$-13\times2^3<0$となるので回復法では減算しないところですが，非回復法では部分余が正なので減算を実行してしまいます．やはり商には‘1’を立てます．その結果，部分余として-80を得ます．次のサイクルでは部分余が負なので，$-80+13\times2^2=-28$という加算を実行し，商には‘-1’を立てます．以下，同じ操作を繰り返して回復法と同一の結果を得ました．

この例では，2サイクル目における減算の誤りが5サイクル目で回復されたことになります．場合によっては最後まで回復されず，余が負で終わってしまうこともあり，これに対しては補正が必要です．これについては，後ほどお話ししましょう．

また，部分商のとる値は$pq=\{-1, 1\}$となり，商は通常の2の補数表現ではないので（BSD：binary signed digitという表現），周囲との整合をとるために変換を施さなくてはなりません．これについても，後で説明します．

● 非回復法の成立条件は負の値を含む

前回，部分余と除数を比較する時点で$pr_i<2d$であることが回復法での成立条件と述べました（正確には$0\leqq pr_i<2d$となる）．

一方，非回復法での成立条件は以下のようになります．

$$-2d<pr_i<2d \quad \cdots\cdots(5)$$

確認してみましょう．

$0\leqq pr_i<2d$の場合は回復法と同じなので説明を省略します．では，$-2d<pr_i<0$の場合はどうでしょう．部分余が負なので加算を行うことになり，式(6)のようになります．

$$pr_{i+1}=pr_i+d>-d \quad \cdots\cdots(6)$$

次のサイクルでシフト・アップしても，$-2d<pr_i<0$は成立するので，式(5)が最初のサイクルで成立していれば，問題なくサイクルを重ねることができます．

この条件をどのように初期化で実現するかは回復法の場合と同じなので，前回を参照してください．

● メリットは符号付き引き数を扱えること

ところで，非回復法のメリットは何でしょうか．

一つは，前サイクルの部分余の符号によって部分商や部分余の決定がなされるということです．つまり，サイクルの先頭で部分商や加算/減算の処理が決まっているわけです．回復法では，部分商や部分余の決定は減算の結果の符合によって判断されていたので，もっとも遅いパスが確定するまで結果が得られませんでした．このことは，1サイクル当たりの遅延を改善することができます．しかし，加算/減算がいらなくなったわけではないので，回路全体から見れば若干の改善といえます．

もっとも大きなメリットは，「符号付きの引き数が扱える」ということです．**図2**で示したように，非回復法では部分余は符号付きで演算されるので，部分余の初期値である被除数も符号付きで扱えることになります．また，除数の加算/減算が使い分けられるということは，除数に対しても符号付きで演算できるといえます（除数が正であるときの加算/減算は，負である場合には減算/加算となるわけなので…）．

ただし，上述したアルゴリズムをそのまま利用するというわけにはいきません．若干のアルゴリズムの修正が必要になります．**図1**のフローチャートは**図3**のように書き換えられます．**図1**と異なるのは，部分余の判定部（あるいは加算/減算の判定部）だけであり，部分余と除数の符号を比較して分岐することになります．演算例を**図4**に示します．

● 非回復法では演算後，二つの後処理が必要

非回復法による除算のコア部分は上述したとおりですが，ほかにも次の二つの後処理を行う必要があります．

- **BSD表現の商を2の補数表現へ変換する処理**
- **余の符号が被除数の符号と異なる状態で除算が終了した場合の補正処理**

まず，BSDを2の補数に変換する方法について説明します．**図4**ではどのようにこの処理を行っているかというと，商の‘-1’となったビットを‘0’に変換し，これをpとして式(7)の演算を行います．

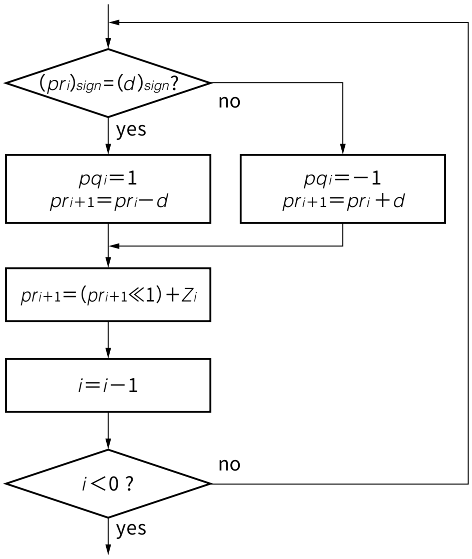


図3　符号付き引き数を扱うための除算アルゴリズム

実際に演算回路を実現するためのアルゴリズム（繰り返し部分のみ）を示す．図1と違うのは部分余の判定部のみ．部分余の正負の判定から部分余と除数の符号が等しいかどうかの判定に変更．

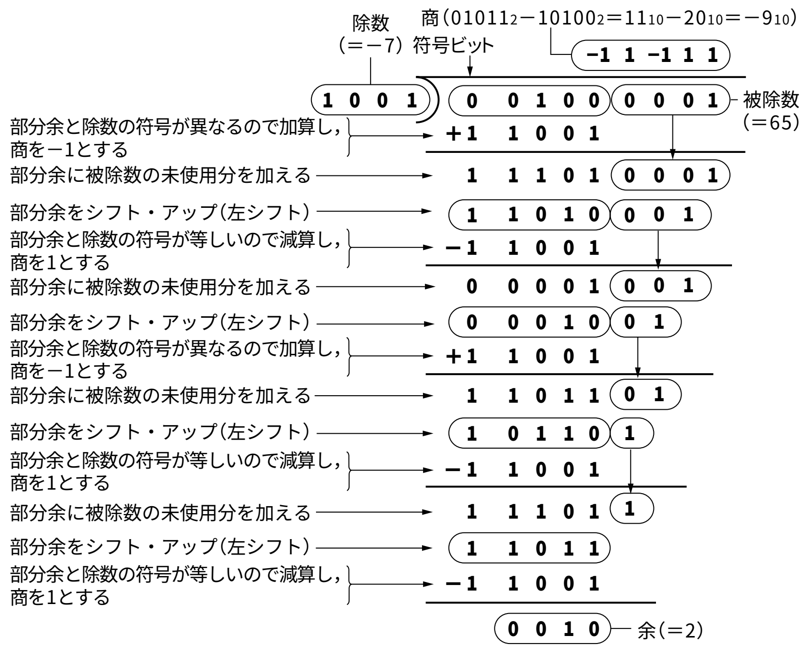


図4　図3のアルゴリズムに沿った演算

この図では，BSDを2の補数に変換するために，商の‘−1’となったビットを‘0’に変換し，$q=p-\overline{p}$ の演算を行っている．

$$q=p-\overline{p} \quad \cdots\cdots(7)$$

$\overline{p}$ は p のビット反転

これはまちがいではないのですが，ここには減算が1回発生しています．できれば余分な演算は避けたいところです．また，式(7)をよく見ると，なにか冗長な感じがします．そこで，まず $-\overline{p}$ を次のように変形します．

$$\begin{aligned}-\overline{p}&=-\sum_{i=0}^{k-1}2^i\,\overline{p_i}\\&=-\sum_{i=0}^{k-1}2^i(1-p_i)\\&=\sum_{i=0}^{k-1}2^i p_i-\sum_{i=0}^{k-1}2^i\\&=p-2^k+1\end{aligned} \quad \cdots\cdots(8)$$

これにより，式(9)が成立します．

$$q=p-\overline{p}=2p-2^k+1 \quad \cdots\cdots(9)$$

ここで，

$$2p+1=\left(p_{k-1},\ p_{k-2},\ \cdots p_1,\ p_0,\ 1\right)_2 \quad \cdots\cdots(10)$$

なので，式(9)における -2^k と式(10)におけるMSB（most significant bit）の p_{k-1} の加算は，式(11)で表されます．

$$2^k p_{k-1}-2^k=2^k\left(p_{k-1}-1\right)=2^k\,\overline{p_{k-1}} \quad \cdots\cdots(11)$$

つまり，

$$q=\left(\overline{p_{k-1}},\ p_{k-2},\ \cdots p_1,\ p_0,\ 1\right)_2 \quad \cdots\cdots(12)$$

と変換され，減算を用いなくても結果が求められます．

したがって，部分商を保存する場合は最初から‘−1’を‘0’として（p に変換された形で）保存しておけばよいことになります．

ところで，式(12)を見ると結果が $k+1$ ビットとなっているのがおわかりでしょう．もともと求めた q は k ビットだったはずなので，MSBの1ビットはあってはなりません．$\overline{p_{k-1}}$ を省略するためには，$\overline{p_{k-1}}=p_{k-2}$ である必要があります．言い換えれば，$q_{k-1}\neq q_{k-2}$ であればよいことになるので，最初のサイクルにおける加減算で部分余の符号が反転すればよいわけです．よって，式(13)が成立すれば条件が満たされます．

$$-d<pr_{initial}<d \quad \cdots\cdots(13)$$

$pr_{initial}$ は，最初のサイクルで加減算にあてられる z（被除数）の上位部分を示します．最初のサイクルにおけるけた

の位置合わせを式(13)を満たすように行えば，kビットの符号付きの商が求められます．これは，式(5)の条件をより厳しくするものであり，非回復法除算の成立条件を阻害するものではありません．

次に，余の符号が被除数の符号と異なる状態で除算が終了した場合ですが，これは以下のように対応します．

この場合，除算そのものは終了しているので，次のサイクルで回復するということができません．そこで別途，補正処理が必要となるわけです．式(5)の成立条件は最終的な余に対しても成立しますが，この段階ではすでにシフト・アップする必要はないので次式が成立します．

$$-d < r < d \quad \cdots\cdots (14)$$

したがって，rに対しdを加算または減算することで（どちらを実行するかはrとdの符号による），rの符号を反転させることができるので，これによって余の符号と被除数の符号をそろえられます．当然，この処理を行った場合，商に対しても相応の補正を加える必要があります．この後処理をまとめると，**表1**のようになります．

表1 余と商の補正

z	r	d	余補正	商補正
正	正	−	補正なし	補正なし
	負	正	$+d$	−1
		負	$-d$	+1
負	正	正	$-d$	+1
		負	$+d$	−1
	負	−	補正なし	補正なし

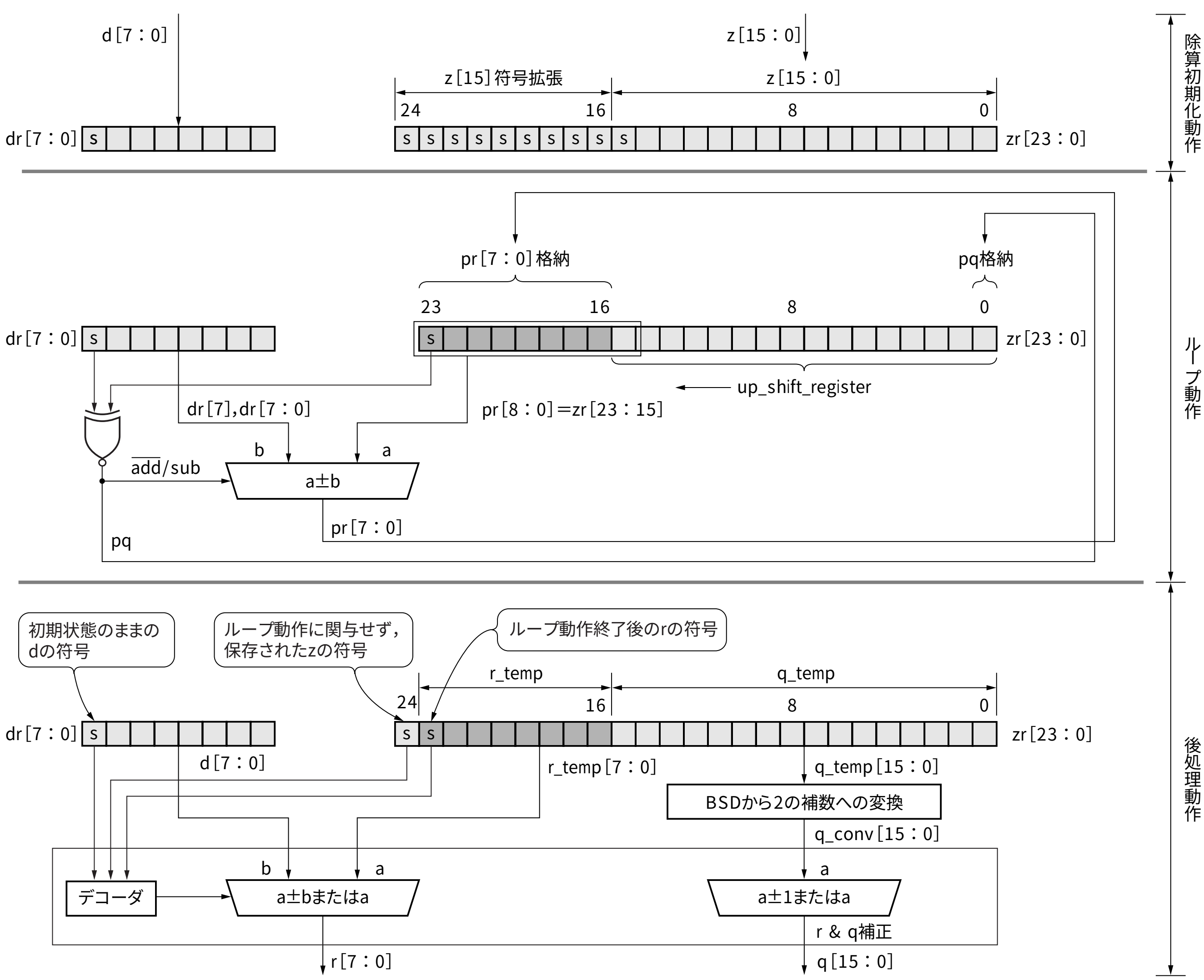


図5 非回復法による除算回路のブロック図
前処理（初期化），ループ処理，後処理の3段階に分けて図示している．なお，この図にはシーケンスにかかわる信号などは表記されていない．

2 非回復法をもとに回路を設計してみる

では，非回復法による除算回路を設計してみましょう．

被除数zを16ビット，除数dを8ビットの符号付き整数とします．$-2d < pr_{initial} < 2d$とするための方法としては，回路を簡略化するため，被除数の上位に符号を拡張することにします．

図5に除算回路のブロック図を示します．前処理（初期化），ループ処理，後処理の3段階に分けて図示しています．動作としては，以下のようになります．

1) re（リセット・イネーブル）でシーケンスを初期化する（$i=0$）．
2) drを除数dの，zrを被除数zの格納レジスタとする．zrは上位に符号拡張を行うため7ビット分余分にけたを取るが，部分余の格納にも流用すること，さらに後処理でzの符号が必要となることから，合計9ビット上位に拡張する．startパルスでz，dがそれぞれのレジスタに格納され，除算の初期化が終わり，$i=16$にセットされる．
3) 前サイクルのpr（zr［23］）とdの符号（dr［7］）を比較してpqが決まり，$pr \pm d$を演算してこのサイクルにおける部分余prが決定する．次のサイクルに移行するにあたり，pq（1ビット）はzrのLSB（least sgnificant bit）に，pr（8ビット）はzr［23：16］に戻される．さらに，zr［14：0］に格納されていた

リスト1　図5の回路構成のソース・コード

```
module          div_nonrestore_a  (
  q     ,  // quotient  [15:00]:signed integer
  r     ,  // remainder [07:00]:signed integer
  busy  ,  // busy

  z     ,  // dividend [15:00]:signed integer
  d     ,  // divisor  [07:00]:signed integer
  start ,  // start (divider trigger)
  clk   ,  // clock
  re  ) ;  // reset enable

// io
  input   [15:00]    z      ;
  input   [07:00]    d      ;
  input              start  ;
  input              clk    ;
  input              re     ;

  output   [15:00]   q      ;
  output   [07:00]   r      ;
  output             busy   ;

// reg
  reg      [04:00]   i      ;
  reg      [24:00]   zr     ;
  reg      [07:00]   dr     ;

// sequence（シーケンサ）
  wire  en_seq = ( i > 0 );

  always  @( posedge clk ) begin
          if  ( re )  i  <= 5'h00 ;
          else if  ( start )  i <= 5'h10 ;
          else if  ( en_seq ) i <= i - 1'b1 ;
    end

// function（演算処理部）
  wire  [08:00]  pr = zr[23:15] ;
  wire           pq = zr[23]~^dr[07] ;
  wire  [08:00]  pr_nxt = ( pq )? pr - {dr[07],dr} :
                                  pr + {dr[07],dr} ;
  wire  [24:00]  zw =
                {zr[24],pr_nxt[07:00],zr[14:00],pq} ;

  always  @( posedge clk ) begin
          if  ( start )        zr  <= {{9{z[15]}},z}  ;
          else if  ( en_seq ) zr  <= zw ;
  end

  always  @( posedge clk ) begin
          if  ( start )        dr  <= d  ;
  end

// final compensation（後処理）
  wire   [15:00] q_temp = zr[15:00] ;
  wire   [07:00] r_temp = zr[23:16] ;
  wire            z_sign = zr[24]  ;

  wire   [15:00]  q_conv = {q_temp[14:00],1'b1} ;

  function       [23:00]  fcompe   ;
          input [15:00]  q_temp   ;
          input [07:00]  r_temp   ;
          input [07:00]  d        ;
          input          z_sign   ;
          casex ({z_sign ,r_temp[07] ,d[07] })
                3'b00x : begin
                        fcompe[23:16] = r_temp ;
                        fcompe[15:00] = q_temp ;
                      end
                3'b010 : begin
                        fcompe[23:16] = r_temp + d ;
                        fcompe[15:00] = q_temp - 1'b1 ;
                      end
                3'b011 : begin
                        fcompe[23:16] = r_temp - d ;
                        fcompe[15:00] = q_temp + 1'b1 ;
                      end
                3'b100 : begin
                        fcompe[23:16] = r_temp - d ;
                        fcompe[15:00] = q_temp + 1'b1 ;
                      end
                3'b101 : begin
                        fcompe[23:16] = r_temp + d ;
                        fcompe[15:00] = q_temp - 1'b1 ;
                      end
                3'b11x : begin
                        fcompe[23:16] = r_temp ;
                        fcompe[15:00] = q_temp ;
                      end
          endcase
  endfunction

  wire   [23:00]  compe   = fcompe (q_conv ,r_temp ,dr ,z_sign) ;

  assign          q       = compe[15:00] ;
  assign          r       = compe[23:16] ;
  assign          busy    = en_seq ;

endmodule
```

前サイクルのシフト・アップ済み部分余（→ pr = zr[23:15]）
前サイクルのpr(zr)とdの符号によってpqが決定（→ pq = zr[23]~^dr[07]）
pqによって加算または減算を実行し，部分余を決定（→ pr_nxt）
次サイクル用zrレジスタ内容（→ zw）
●zr［24］：オリジナルのzの符号
●pr_nxt［07：00］：シフト・アップ済み部分余
●zr［14：00］：シフト・アップ済み被除数の未使用部
●pq：部分商
BSDから2の補数に変換（→ q_conv）
表1による余と商の補正関数（→ fcompe）

未使用のzは1ビットだけシフト・アップしてzr[14]は次のprのLSBとなる．$i=i-1$が実行される．

4）3)のサイクルを繰り返すことで，zr上のzのデータは上位から1ビットずつ消化され，zr上から消えていく．代わりに，zrのLSBに順次格納されたpqがzのシフト・アップとともに上位へシフトされ，zrの下位側に蓄積される．

5）$i=0$となったところでループ動作は終了となる．zr[23：16]には余(r_temp)が，zr[15：0]には商(q_temp)が残り，zはすべてレジスタ上からなくなる．

6）後処理として，q_temp(BSD)を2の補数に変換し，zの符号(z[24])とr_tempの符号を比較して，商と余に補正をかけて，除算を終了する．

この回路構成によるVerilog HDLのRTL記述を，**リスト1**に示します．注意すべき点は，$pr\pm d$を演算する部分でしょう．機能をわかりやすく実現するため**リスト1**のような記述としていますが，論理合成の条件によっては，加算器と減算器がそれぞれ推定されてその結果をマルチプレクスする，という結果になってしまう場合があります．回路的には加減算器として構成することが可能なので，それを適用するように考えなければなりません．回路規模や遅延についてもこちらが有利となります．このループ処理部分は除算器の性能を決定する部分なので，遅延には気を使いましょう．

● ある条件下で演算エラーを起こすという欠陥

長々と説明しておきながらたいへん申しわけないのですが，実は上述のアルゴリズムには欠陥があるのです．特定の条件のときに演算エラーを起こします．その条件は以下のとおりです．

表2　余と商の補正(修正済み)

z	r	d	r=0	r=d	r=−d	余補正	商補正
−	−	−	○	×	×	補正なし	補正なし
−	−	−	×	○	×	−d	＋1
−	−	−	×	×	○	＋d	−1
正	正	−	×	×	×	補正なし	補正なし
正	負	正	×	×	×	＋d	−1
正	負	負	×	×	×	−d	＋1
負	正	正	×	×	×	−d	＋1
負	正	負	×	×	×	＋d	−1
負	負	−	×	×	×	補正なし	補正なし

- 被除数zの符号が負の場合，最後のサイクルで割り切れると演算エラーを起こす
- 除算途中のサイクルで完全に割り切れた場合，演算エラーを起こす

最初の事例から検証しましょう．最後のサイクルで割り切れた場合，$r=0$となります．rの符号は正ですが，このときにzが負であると，**表1**に従い補正がかかってしまうのです．割り切れるということは，補正は必要ない状態なので，ここで補正を施してはなりません．つまり，**表1**の条件が不完全であり，修正が必要です．

また2番目の事例は，以下のように考えられます．

除算途中のサイクルで完全に割り切れた場合は$pr=0$となるので，次のサイクルでシフト・アップしてもやはり0です．これに対して加算/減算のいずれかがかならず実行されるため，このサイクルでは$pr=\pm d$となります．さらに次のサイクルにおけるシフト・アップ後には$pr=\pm 2d$となってしまうため，式(5)の成立条件を満たさなくなってしまいます．この時点で$pr<0$となるのは明らかなので，かりに部分余が'0'であってもzの未使用のけたが'0'でなければ，いずれ式(5)の条件に復帰できます．しかし，zの未使用のけたも'0'となって完全に割り切れてしまった場合は最後まで復帰できず，$r=\pm d$でループ処理を終了します．符号がzと等しい場合は補正もかからないので，そのまま除算を終了してしまうわけです．

これらの演算エラーに対する対処法は，いくつか考えられます．

- **部分余が0になった場合，加算/減算を行わない**――この場合，商のビットがとりうる値はpq={−1，0，1}となり，pqを保存するレジスタの構造を変更する必要がある．また，BSDからバイナリへの変換が先の説明のような簡単な方法ではなくなる．
- **サイクルの途中で完全に割り切れた場合，シーケンスを抜けて除算終了とする(補正不要)**――この場合，zとqが同一レジスタ上に混在している**図5**のような構造では，どこまでがzの未使用部分であるかの判定がめんどうになる．
- **$r=\pm d$でループ処理を終了した場合，後処理で補正をかける**――この場合，後処理の構造が複雑になる．

ループ処理部分はできる限りシンプルにしあげておくことが性能面で得策であると判断し，ここでは最後の対処法を採用することとしましょう．最後のサイクルで割り切れた場合の対処も含め，**表1**を**表2**のように修正します．

コラム　除算と浮動小数点表現

除算は四則演算と総称される演算の一つですが，この4種類の演算にはそれぞれ個性があります．

演算回路の構成としては，加減算，乗算，除算の3種類に分類できますが，けたの管理という観点では加減算，乗除算の2種類に分類できそうです．浮動小数点演算の場合が典型ですが，乗除算は指数部と仮数部を独立に演算することができるのに対し，加減算ではけた位置を合わせてから演算を行わなくてはならず，指数部と仮数部は不可分と言えます．

非回復法における除算の成立条件$-2d < pr_i < 2d$を満たすため，除算を始めるに際に初期化が必要になるということは本文中で述べたとおりです(回復法についても同じ)．回路の設計例では，単純化するために上位に符号を拡張することで済ませていますが，それによってむだなサイクルが発生することは明らかです．ここで，先に述べた「乗除算ではけたの管理は分離できる」という特質を生かして演算システムが浮動小数点化されていれば，初期化の操作は不要となり，むだなサイクルの発生も防ぐことができます．「除算は浮動小数点と相性が良い演算である」と言えるでしょう．

システムの演算形式をこの理由だけで決めることはできませんが，いろいろな可能性を探ってみるのも悪くはないと思います．

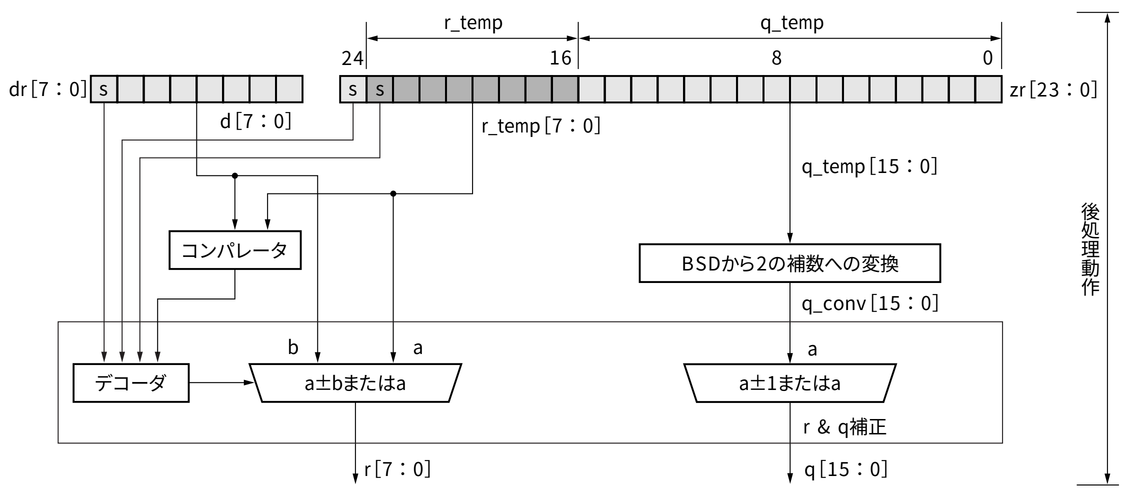


図6　修正を施した除算回路のブロック図
表2の補正を採用することで，r_tempと±d，0とのコンパレータが追加され，補正制御用のデコーダが複雑化する．

図6に修正を施した非回復法における除算回路のブロック図を，p.144の**リスト2**にそのRTL記述を示します．なお，ブロック図は後処理部分のみ示します．

以上で非回復法による除算器が完成しました．ループ処理部分は加減算を行えるようにしたことで逆にシンプルになり，性能の向上が期待できます．ただし，後処理が複雑になってしまったのが残念なところです．では，入力段で引き数を符号なしとし，回復法を用いて最終段で符号を戻す，という構造と比べてどうでしょうか．判断はみなさんにお任せしますが，ここまでの説明で「符号付きでも除算ができる」ということと，「意外にめんどうではない」ということが理解していただけたかと思います．

＊　　＊　　＊

今回の非回復法を用いて符号付きの引き数を直接使用して除算を行えるようになりました．次回は，ここまでを土台として，本格的に性能の向上を目指します．

すずき・しょうじ
スタンダード・ロジック(株)

<筆者プロフィール>
先日，本誌の原稿料で手に入れたロード・レーサーで，梅池サイクル2005(ひたすら登るだけの自転車レース)に参加しましたが，あまりの遅さにがっくり．心機一転ガンバロウと思ったら梅雨入りで，ブルーな毎日です．

リスト2　図6の回路構成のソース・コード

```
module         div_nonrestore_b   (
  q      ,  // quotient  [15:00]:signed integer
  r      ,  // remainder [07:00]:signed integer
  busy   ,  // busy

  z      ,  // dividend [15:00]:signed integer
  d      ,  // divisor  [07:00]:signed integer
  start ,  // start (divider trigger)
  clk    ,  // clock
  re     ); // reset enable

// io
  input  [15:00]  z      ;
  input  [07:00]  d      ;
  input           start ;
  input           clk    ;
  input           re     ;

  output [15:00]  q      ;
  output [07:00]  r      ;
  output          busy   ;

// reg
  reg    [04:00]  i      ;
  reg    [24:00]  zr     ;
  reg    [07:00]  dr     ;

// sequence（シーケンサ）
  wire  en_seq = ( i > 0 ) ;

  always @( posedge clk ) begin
         if ( re ) i <= 5'h00 ;
         else if ( start )  i <= 5'h10 ;
         else if ( en_seq ) i <= i - 1'b1 ;
  end

// function（演算処理部）
  wire   [08:00]  pr = zr[23:15] ;
  wire            pq = zr[23]~^dr[07] ;
  wire   [08:00]  pr_nxt = ( pq )? pr - {dr[07],dr}
                                 :pr + {dr[07],dr} ;
  wire   [24:00]  zw = {zr[24], pr_nxt[07:00],
                                  zr[14:00], pq} ;

  always @( posedge clk ) begin
         if ( start ) zr <= {{9{z[15]}},z} ;
         else if ( en_seq ) zr <= zw ;
  end

  always @( posedge clk ) begin
         if ( start ) dr <= d ;
  end
```

前サイクルのpr(zr)とdの符号によってpqが決定

前サイクルのシフト・アップ済み部分余

pqによって加算または減算を実行し，部分余を決定

次サイクル用zrレジスタ内容
- zr[24]：オリジナルのzの符号
- pr_nxt[07：00]：シフト・アップ済み部分余
- zr[14：00]：シフト・アップ済み被除数の未使用部
- pq：部分商

```
// final compensation（後処理）
  wire   [15:00]  q_temp = zr[15:00] ;
  wire   [07:00]  r_temp = zr[23:16] ;
  wire            z_sign = zr[24] ;

  wire   [15:00]  q_conv = {q_temp[14:00],1'b1} ;
  wire            r_non0 = ( r_temp!=8'h00 ) ;
  wire   [07:00]  dr_cmp = ~dr + 1'b1 ;
  wire            req_pd = ( r_temp== dr     ) ;
  wire            req_nd = ( r_temp== dr_cmp ) ;

  function        [23:00]  fcompe ;
          input   [15:00]  q_temp ;
          input   [07:00]  r_temp ;
          input   [07:00]  d      ;
          input            z_sign ;
          input            r_non0 ;
          input            req_pd ;
          input            req_nd ;
          casex   ({r_non0 ,req_pd ,req_nd ,z_sign ,r_temp[07],
                                                   d[07] })
                  6'b0xxxxx : begin
                                fcompe[23:16] = r_temp ;
                                fcompe[15:00] = q_temp ;
                            end
                  6'b11xxxx : begin
                                fcompe[23:16] = r_temp - d ;
                                fcompe[15:00] = q_temp + 1'b1 ;
                            end
                  6'b1x1xxx : begin
                                fcompe[23:16] = r_temp + d ;
                                fcompe[15:00] = q_temp - 1'b1 ;
                            end
                  6'b10000x : begin
                                fcompe[23:16] = r_temp ;
                                fcompe[15:00] = q_temp ;
                            end
                  6'b100010 : begin
                                fcompe[23:16] = r_temp + d ;
                                fcompe[15:00] = q_temp - 1'b1 ;
                            end
                  6'b100011 : begin
                                fcompe[23:16] = r_temp - d ;
                                fcompe[15:00] = q_temp + 1'b1 ;
                            end
                  6'b100100 : begin
                                fcompe[23:16] = r_temp - d ;
                                fcompe[15:00] = q_temp + 1'b1 ;
                            end
                  6'b100101 : begin
                                fcompe[23:16] = r_temp + d ;
                                fcompe[15:00] = q_temp - 1'b1 ;
                            end
                  6'b10011x : begin
                                fcompe[23:16] = r_temp ;
                                fcompe[15:00] = q_temp ;
                            end
          endcase
  endfunction

  wire   [23:00]  compe = fcompe (q_conv ,r_temp ,dr ,z_sign ,
                                r_non0 ,req_pd ,req_nd ) ;

  assign          q     = compe[15:00] ;
  assign          r     = compe[23:16] ;
  assign          busy  = en_seq ;

endmodule
```

BSDから2の補数に変換

r≠0を検出

dの2の補数を演算

r=dを検出

r=−dを検出

表2による余と商の補正関数

C&E基礎解説シリーズ　好評発売中

最新わかる半導体

不思議な半導体の働きとミクロの世界で動くLSI

傳田 精一 著　A5判 216ページ　定価1,680円(税込)　ISBN4-7898-3628-2

CQ出版社　〒170-8461 東京都豊島区巣鴨1-14-2 販売部 ☎ (03)5395-2141 振替 00100-7-10665